Approximate Subtree Search for Labeled Trees with Small Depth Value

László Kovács, Tibor Répási, Erika Baksa-Varga

Department of Information Technology, University of Miskolc
kovacs@iit.uni-miskolc.hu
repasi@iit.uni-miskolc.hu
iitev@uni-miskolc.hu

Abstract: In many scientific areas there is a frequent need to extract a common pattern from multiple data. In most cases, however, an approximate but low cost solution is preferred to a high cost exact match. To establish a fast search engine an efficient heuristic method should be implemented. Our investigation is devoted to the approximate nearest neighbor search (ANN) for unordered labeled trees. The proposed modified best-first algorithm provides a $O((Nq+Nb)\cdot M+K\cdot Nq\cdot Nb/M)$ cost function with simple implementation details. According to our test results, realized with smaller trees where the brute-force algorithm could be tested, the yielded results are a good approximation of the global optimum values.

Keywords: tree matching, approximate nearest neighbor search

1 Introduction

In many scientific areas there is a frequent need to extract a common pattern from multiple data. The most common structure of the data is a hierarchy or tree. The task is to determine the set of sub-trees having the best matching with the pattern. One of the important application areas for sub-tree matching is the area of information systems, where the most common new storage format is the XML tree. XML is seeing increased use and promises to fuel even more applications in the future. An XML document can be modeled as a tree. Each node in this tree corresponds to an element in the document. Each edge represents inclusion of the element corresponding to the child node under the element corresponding to the parent node in the XML file.

A significant trend in data management is to store the database in XML format. A common characteristic of databases is that the created data tree is relatively wide and relatively shallow. Taking for example an insurance company, the number of customers and the number of contracts or the number of insured objects can be several thousands or millions. On the other hand, the number of encapsulation levels in the data structure for database objects may be less than 10. Thus the depth of the database tree may be about 10 and the number of leaves may be some millions. The goal of our investigation is to find a method for approximate subtree matching which is suited for this kind of trees.

In these applications the nodes of a tree are characterized by one or more attributes. The description vector of the nodes is called the label of the nodes. Focusing on database applications, the label of a node contains always two elements: a type and a value description. Regarding the type component, the tree should meet a schema constraint. The constraints for value component may be given by integrity rules. In some areas, not only the node types but also the order of nodes is important. In this case, the children are assigned to an ordering number in the scope of the parent. XML documents, for instance have an ordered and labeled tree structure. In our investigation we are focusing on unordered labeled tree structures. A recent workshop report from Yale suggested that more research should be undertaken to improve the heuristic search using algorithms designed to meet the demand made by increasingly large tree datasets [1].

Reviewing [2] the different researched approaches to comparing trees, a great number of algorithms have been developed so far to solve these problems. P. Bille published an extensive survey [3] on comparing trees with exact searching methods. As a conclusion from his work it turned out that all of the unordered versions of the problems in general are NP-hard. Indeed, the tree edit distance and alignment distance problems are even MAX SNP-hard. However, using special constraints polynomial time algorithms are available, just like for the ordered versions of the problems. These are all based on the classic technique of dynamic programming. Also, a large amount of work has been performed for comparing unordered trees based on various distance measures, especially on edit distance as the most commonly used distance measure. Shasha et al. [4], however, proposed a new approach, called *Atree-Grep*. They addressed the *approximate nearest neighbor search* problem for unordered labeled trees. Their algorithm, called '*pathfix*', consists of two phases. First, the paths of the trees are stored in a suffix array and then the number of mismatching paths are counted between the query tree and the data tree. To speedup the search, they use a hash-based technique to filter out unqualified data trees at an early stage of the search. The algorithm has been implemented into two special Web-based search engines and proved to be fast, particularly when the dictionary size of node labels is large.

2 Distance Measures for Tree Comparison

As could be seen from the review of the researches in tree comparison, most of the proposals in subtree matching are based on the edit distance between trees. This distance metric is a natural extension of the edit distance concept used for string comparisons. This metric provides an exact distance measurement between the trees. The drawback of these algorithms is the high cost of the computations. In the case of online applications with large tree datasets, the execution time is a crucial factor. In the case of edit distance, a set of elementary transformation functions is defined on the T_D set of trees. This set is denoted as E_D. The cost value of the elementary transformations is a non-negative real number. The corresponding cost function is denoted by

$c : E_D \rightarrow R^+$.

It is assumed that T_D is closed to E_D, i.e.

$\forall e \in E_D \; : \; e : T_D \rightarrow T_D,$

$\forall T_1, T_2 \in T_D : \exists e_1, e_2, \ldots, e_m \in E_D : e(T_1) = e_m \, o \, e_{m-1} \, o \ldots e_2 \, o \, e_1(T_1) = T_2.$

Let us denote the set of chain of transformations from T_i to T_j by $E_{i,j}$. The cost of chain e is defined as the sum of the single transformation steps:

$c(e) = \Sigma\, c(e_i)$.

The edit distance between T_i and T_j is defined as the minimal cost of transformation chains from T_i to T_j:

$c_{i,j} = min\{ c(e) \mid e \in E_{i,j} \}$.

Usually, the following elementary e operations are defined for tree objects:

- *relabel*: assigns a new node name to the root of the tree
- *insert*: inserting a new node into the children of the root node
- *delete*: deleting a node from the children of the root node
- *insert tree*: inserting a tree under the root node
- *delete tree*: deleting a tree from the children of the node

The list of elementary transformations with minimal cost is usually generated with a dynamic programming method. According to [3, pp.7], the tree distance value can be calculated using the following recursive formula:

$$\begin{aligned} d(0,0) &= 0 \\ d(F,0) &= d(F-v,0) + c(v,0) \\ d(0,F) &= d(0,F-v) + c(0,v) \\ d(F_1,F_2) &= min \begin{cases} d(F_1-v,F_2) + c(T(v),0) \\ d(F_1,F_2-v) + c(0,T(v)) \\ d(F_1-T(v),F_2- T(w)) + c(T(v),T(w)) \end{cases} \end{aligned}$$

where *F* denotes a tree and *T(v)* denotes a tree with root element *v*. The computation cost of the basic dynamic programming method for trees is O(|T|4). It is proved in [4] that the ANN problem for edit distance metric is an NP-complete problem. In spite of this difficulty, most of the proposals for ANN searching for trees use the edit distance measure. There are very few proposals that apply a simplified distance function to provide a lower cost solution. A good example for this approach is [4], where the distance from T_1 to T_2 is measured with the total number of root-to-leaf paths in T_1 that do not appear in T_2. The nodes in T_2 that do not appear in T_1 can be freely removed.

3 The Proposed Matching Algorithm

To provide a more efficient dissimilarity computation, a modified distance measure was created. During the editing process every vertex of the query tree is either transformed into a vertex of the base tree or is deleted. Based on this transformation, every vertex of the query tree can be mapped either to a target vertex or to the sink symbol. Using this approach, a generalized mapping can be defined between the query and the base tree. We define *m( )* as the distance mapping from T_1 to T_2 in the following way:

1. $m : V(T_1) \rightarrow V(T_2) \cup \varepsilon$
2. $\forall v, m(v) \in V(T_2) : l(v) = l(m(v))$
3. $\forall v_1 \neq v_2, m(v_1), m(v_2) \in V(T_2) : m(v_1) \neq m(v_2)$
4. $\forall v_1 \neq v_2, m(v_1), m(v_2) \in V(T_2): v_1 < v_2 \Leftrightarrow m(v_1) < m(v_2)$

The fourth property is called ancestor condition, the ancestor-descendants relationship among the query vertices must be preserved in the target tree, too. Other types of relationships among the query vertices are neglected and not preserved. The parent-child relationships are the only important information stored in the query tree. The absence of an edge means in our approach a 'do not know' information. In this case, we do not care about the existence of an edge between the mapped vertices in the base tree. Figure 1 shows an example for this mapping.

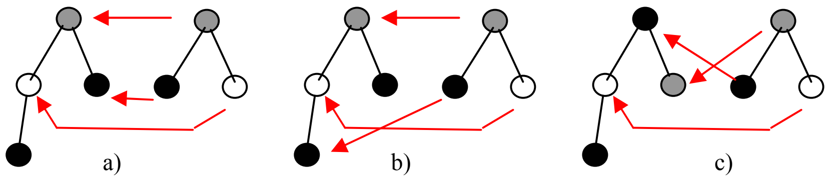


Figure 1

Figure 1a) and Figure 1b) show valid mappings. The sibling nodes in the query tree are mapped to sibling nodes in Figure 1a), and to parent-child nodes in Figure

1b). Figure 1c) shows an invalid mapping as the parent-child relationship is not preserved. Based on this mapping, a distance value can be defined between two trees. The cost of mapping *m* is defined as the sum of the vertex mappings related to the query tree:

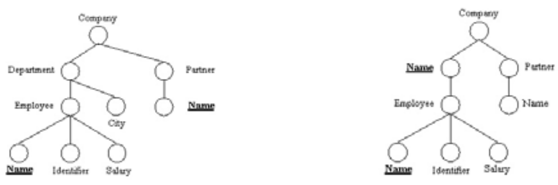
$$cost(m) = \Sigma_{n \in V(T)}\, c(n),$$

where

$$c(n) \quad = \quad \begin{cases} C_2, \text{ if } m(n) = \varepsilon \ \vee \ m(r(T)) = \varepsilon \\ 0, \text{ if } n = r(T) \ \wedge \ m(r(T)) \neq \varepsilon \\ C_1 \ \ (d(m(n),m(pp(n)) - 1) \text{ otherwise.} \end{cases}$$

In this definition, *pp(n)* denotes the nearest ancestor of *n* in the query tree which is mapped to a *non-ε* element. If the root of the query tree is mapped to *ε* then *c(n)* is C_2, otherwise the path from *n* to *r(T)* (excluding *n* and including *r(T)*) contains minimum one vertex mapped to a *non-ε* value. In this case both *m(n)* and *m(pp(n))* are *non-ε* elements. The *d( )* function denotes the length of path from *m(pp(n))* – *m(n)* in the base tree. As mapping *m* preserves the parent-child relationship, *m(pp(n))* is an ancestor of *m(n)*. Thus *d( )* yields a positive integer value. C_1 and C_2 are cost units. C_1 corresponds to gap-lengths between two preserved vertices and C_2 denotes the cost for vertex deletion. In our approach, C_2 is greater than C_1 since the absence of an element means a larger difference than the relocation of the element.

In [2] we introduced a modified best first algorithm that works on a state-tree, the nodes of which are assigned not to the vertices but to the vertex mappings of the query tree. In this paper, the proposed method is designed to work with database trees where the trees have a small depth value and a large width value. Another assumption is that the number of node-types is low but the number of the different node-values is large as a database contains usually several thousand instances of the same element type. Another characteristic of the data tree is that the same type can not occur more than once in neither schema paths. This requirement is usually met in the applications since the usage of recursive structures is not allowed in most systems, or at least it makes the structure too complex for users. Figure 2a) shows a valid while Figure 2b) shows an invalid schema.



a) b)

Figure 2

A query tree usually contains some values to be matched in the data-tree. Taking a query to retrieve the departments located in London and having employees with a 1000 Euro salary, the query tree is given by the following tree:

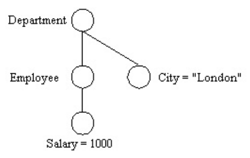


Figure 3

The query tree contains two nodes with type and value pairs (Salary=1000 and City="London"). The result of the query is the set of department nodes having the given descendant nodes. The number of nodes with type 'Salary' may be very large, but the number of nodes with value 'Salary=1000' is significantly lower. The query search engine should detect paths between the matched nodes. These paths in the data-tree should correspond to the ancestor-descendant relationships given in the query tree. It is clear that, it is better to discover these paths from the leaves towards the root than in reverse direction. A node can have only one parent but may have a large number of children. Of course, the bottom-up direction can only be selected if the matching descendant nodes are discovered first. In the proposed algorithm, an index structure is generated to find the matching nodes in an efficient way.

Traditional database management systems usually use B-tree indexes to provide fast access to object instances. The cost of index search is in general O(log(N)). A similar index structure for the *T* data tree is generated in the preparation phase. According to [5], this work can be performed with O(N log(N)) computational cost. In the searching phase a *Q* query tree is given. The approximate matching algorithm first selects the nodes in *T* matching the nodes in *Q*. Matching is performed using the generated index structure. The query tree is transformed into a list of query strings. A query string is a path in the query tree from a leaf to the root. The number of query strings is equal to the number of leaf nodes in the query tree. The query strings for the example query are the followings:

Department; City="London"

Department; Employee; Salary=1000

In the next phase, the matching nodes are processed in decreasing depth order related to the query tree. In the example, the nodes are processed in the following order:

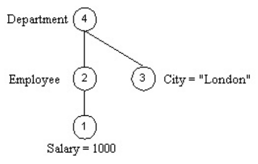


Figure 4

For every node the path to the relative root is discovered. A node is a relative root if it is an ancestor and its type is equal to the type of the root in the query tree. For every discovered relative root nodes an administration vector is created. The elements of this vector correspond to the query strings of the query tree. The administration vector stores the current minimal mapping costs for the query strings.

During node processing, the path to the relative root is compared to each query strings. The cost of mapping is calculated and if it is lower than the current minimum value, the administration vector is updated. The comparison process can be stopped if the current sub-cost is higher than or equal to the current minimum value. At the end of path evaluation in the data tree, the selected nodes of this path are deselected. In the next iteration only the selected nodes are processed. After processing all the selected nodes, the best matching costs for relative root nodes are calculated from the corresponding administration vectors. In the final step, the relative root nodes are ranked based on the calculated minimal cost value. The generated list contains all approximation sub-trees having a distance less than a given threshold.

The cost of the proposed method can be estimated on the following way. In the first step, data nodes related to the nodes of the query tree are selected. Let us use the following denotations:

N: the number of nodes in the data tree,

M: the number of nodes in the query tree,

K: the average number of instances for a value,

L: the average number of instances for a type.

Using the pre-generated index structure, the node selection phase requires

$O(M\ (\ \log(N) + K) + \log(N) + L)$

cost. The number of the selected data nodes is O(M K). The cost of path evaluation for a selected node is O(M) as the length of the paths is O(1). The cost for best matching can be calculated from the administration vectors with O(M L) cost. Thus the cost of the second phase is

$O(\ M^2\ K + M\ L).$

The ranking cost is

$O(L \log\ (L))$

if all approximation subtrees are requested in the query. One can assume that

$N >> L >> K > M$

holds. Thus the total cost can be estimated by

$O(M \log(N) + M^2K + ML + L \log(L)).$

4 Test Results

To be able to test the algorithm we created a test implementation by using SciLab [6] and its extension library MatNet for modeling graphs. Each node of the graph has a label consisting of a type, value pair. Considering the possibilities of Scilab's [6] MatNet we used the node attributes *node_color* and *node_diam* as attribute type and value. The visualization of the graph shows the node types by different colors and its values by different node diameters. For testing we constructed a data tree shown on Figure 5 and a query tree shown on Figure 6. Matching nodes are highlighted on both figures.

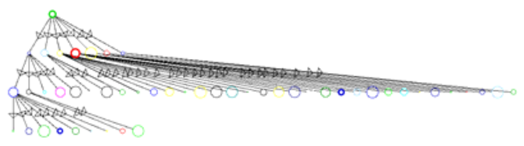


Figure 5

The implementation is using the following workflow: i) Decomposition of a given query tree to its strings. ii) Indexing the data tree by collecting and ordering the node identifier, node type (*node_color*) and value (*node_diam*). Ordering by node type and value is done ascending. Next iii) the program is searching in the data tree for the successors of all strings from the query tree by using binary search algorithm [5] on the previously constructed index structure. Cost calculation is based on the alignment distance of each query string to the date tree. We defined two cost factors, one for the case when a node in the query string is not a predecessor of the corresponding path in the data tree, we call this CQDROP. The other cost factor CHOP is for the inverse case, when a node on the data path is not in the corresponding query string. The alignment cost is a sum of all occasions of cost factors, summarized for all corresponding query strings and data path pairs. The test program is iv) calculating the alignment cost based on the number of data paths corresponding to the query strings.

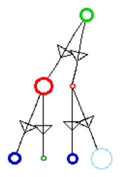


Figure 6

The calculation of the alignment cost of the query tree on Figure 6 is performed to the data tree on Figure 5. Considering the decomposition of the query tree to four strings, beginning from the leafs – from left to right – we can see that the first string can be completely aligned to the data tree, i.e. the alignment cost for this string is zero. For the second string, which cannot be found in the data tree we define a maximal cost CMAX. On the third string we can find the leaf and the head node of the query string in the data tree, but the node in the middle is not found, therefore we add a cost 1*CQDROP. Since the corresponding path in the data tree contains two more nodes which cannot be found in the query string we add 2*CHOP to the cost which will be for this query string 2CHOP+CQDROP. In the case for the last query the same occurs as for the second string: it cannot be found in the data tree, therefore a maximal cost CMAX is added. Thus the total cost of alignment will be C=0+CMAX+(2*CHOP+CQDROP)+CMAX.

A future work will show the computational costs used by the test implementation.

References

[1] J. Cracraft, M. Donoghue: Assembling the tree of life – Research needs in phylogenetics and phyloinformatics, Report from NSF Workshop, Yale University, July 2000

[2] L. Kovács, T. Répási, E. Baksa-Varga: Overview of Nearest Neighbor Subtree Search Methods, in Proceedings of the 5th International Symposium of Hungarian Researchers on Computational Intelligence, ISBN 963 7154 345 pp. 301-312, 2004

[3] P. Bille: Tree Edit Distance, Alignment Distance and Inclusion, IT University of Copenhagen, Technical Report Series TR-2003-23, ISSN 1660-6100, March 2003

[4] Shasha et al.: AtreeGrep – Approximate Searching in Unordered Trees, in Proceedings of SSDBM 2002, Edinburgh, July 2002, pp. 89-98

[5] D. E. Knuth: The Art of Computer Programming – Sorting and Searching, volume 3, Addison Wesley, 2nd edition, Feb. 1998

[6] SciLab Project Page, www.scilab.org